Overview of IA-64 Architecture

This document is part of the HP-UX 11.x Software Transition Kit

Table of Contents

Overview of IA-64 Architecture

Background

Traditional microprocessor architectures have attributes that fundamentally limit their performance. To achieve higher performance, these architectures depend not only on performing instructions faster, but also on performing more instructions per clock cycle. Executing more instructions per clock cycle (referred to as instruction level parallel execution, or parallel execution) allows more information to be processed at one time, which increases overall processor performance.

Today's compilers have limited, indirect control of parallel hardware. Performance is limited both by the compiler's ability to expose parallelism to the processor and by the processor's ability to locate instructions that can be executed in parallel from the sequential stream of instructions produced the compiler. Performance is also limited by program control flow (branches) and memory latency (the time for data to arrive from memory), both of which cause processors to stall and lose valuable information processing time.

Overcoming these limitations requires a new architecture. Intel and Hewlett-Packard have jointly defined a new architecture technology called EPIC (Explicitly Parallel Instruction Computing), named for the ability of a compiler to extract maximum parallelism (potential to do work in parallel) in source code and "explicitly" describe that parallelism to the hardware. Intel and HP have also jointly defined a new 64-bit instruction set architecture (ISA), based on EPIC technology, which Intel has incorporated into IA-64, Intel's 64-bit microprocessor architecture. Now, with the new 64-bit ISA, a compiler can expose, enhance, and exploit parallelism in a program and make it explicit to the hardware. In addition, the new 64-bit ISA includes predication and speculation techniques, which address performance losses due to control flow and memory latency. This innovative approach of combining explicit parallelism with speculation and predication will allow IA-64 to progress well beyond the performance limitations of traditional architectures.

Inherent Scalability

One of the key objectives for the new ISA was to provide a family of microprocessors that balance different performance and cost requirements. The architectural features and instruction format of IA-64 provide transparent compatibility across a spectrum of microprocessors. As the IA-64 product family evolves, additional functional units (the part of a processor that performs calculations) and other processor resources can be added to increase the "width" of a machine. Increasing the width of a machine increases the number of instructions that can be executed in parallel, which boosts performance. This "inherent scalability" was designed into the IA-64 architecture from the beginning. Inherent scalability coupled with explicit parallelism, predication, and speculation will enable significant performance gains.

Explicit Parallelism

To illustrate the limitations of traditional architectures and the benefits of the new 64-bit ISA, think of a processor as operating similarly to a bank lobby. Customers queue up in a single line for various services: loans, deposits and withdrawals, and new accounts. Each service has a teller. At the front of the line, a "greeter" directs customers to the appropriate teller. The greeter can reorder the line to ensure that the tellers are never idle.

However, let's assume that the greeter can reorder only the first five customers in the line. Thus, a particular teller may be idle, because even though customers in the line may need the teller's services, the customers may be too far back in the line. This is analogous to how today's architectures work. The processor receives a sequential stream of instructions from a compiler and must reorder the instructions to prevent functional units from being idle. The processor can only reorder a small, fixed number of instructions. A particular functional unit may be idle even though there are instructions in the instruction stream destined for that functional unit.

Now, assume that the bank lobby has separate lines for each service (teller) and that the greeter goes outside of the back and directs arriving customers to the correct line. Further, assume that the greeter can even phone customers at home and schedule an appointment with the bank. Thus, the greeter can explicitly reorder all of the bank customers, regardless of how many there are, before they enter the bank. Now a teller is idle only if no customer wants that particular service. That's the concept behind explicit parallelism: instructions arrive at the processor explicitly ordered by the compiler. The compiler organizes the code for an entire program and makes the ordering explicit so the processor can execute instructions in the most efficient manner.

Predication

Another major performance limiter for traditional architectures is branches. A branch is a decision between two sets of instructions. In our bank example, the choice between filling out a withdrawal or deposit slip is a branch. Assume, in an effort to respond to customers quickly, that a bank teller wishes to prepare the appropriate deposit or withdrawal slip in advance for the next customer. Since most customers make deposits, the teller predicts that a deposit slip will be needed. When he's right, the deposit is completed quickly; when he's wrong, any customers in line must wait while he completes a withdrawal slip.

Similarly, today's architectures use a method called branch prediction to predict which set of instructions to load. When branches are mispredicted (like when the teller predicted the customer wanted to make a deposit, when they really wanted to withdraw money) the whole path suffers a time delay. While current architectures may only mispredict 5-10% of the time, the penalties may slow down the processor by as much as 30-40%. Branches also constrain compiler efficiency and under-utilize the capabilities of the microprocessor.

The new 64-bit ISA uses a concept called predication. Assume the teller develops a better strategy for responding quickly to customers in line. When he has a spare moment, he prepares both a deposit and withdrawal slip for each customer. Then he uses the appropriate slip and discards the other. The teller now works more efficiently without causing the customers in line to wait. This is similar to predication. The predicates allow both sets of instructions to be executed, but only those that are needed actually get used. Predication can remove many branches and reduce mispredicts significantly. A study in ISCA '95 by S. Malhlke, et. al., demonstrated that predication removed over 50% of the branches and 40% of the mispredicted branches from several popular benchmark programs. Thus, predication enables increased performance resulting from greater parallelism and better utilization of an IA-64 based processor's performance capabilities.

Speculation

Memory latency (the time to retrieve data from memory) is yet another performance limitation for traditional architectures. If memory latency were a bank operation, it would be analogous to opening a new account: it takes a relatively long time. When new customers open accounts, they hold up the entire

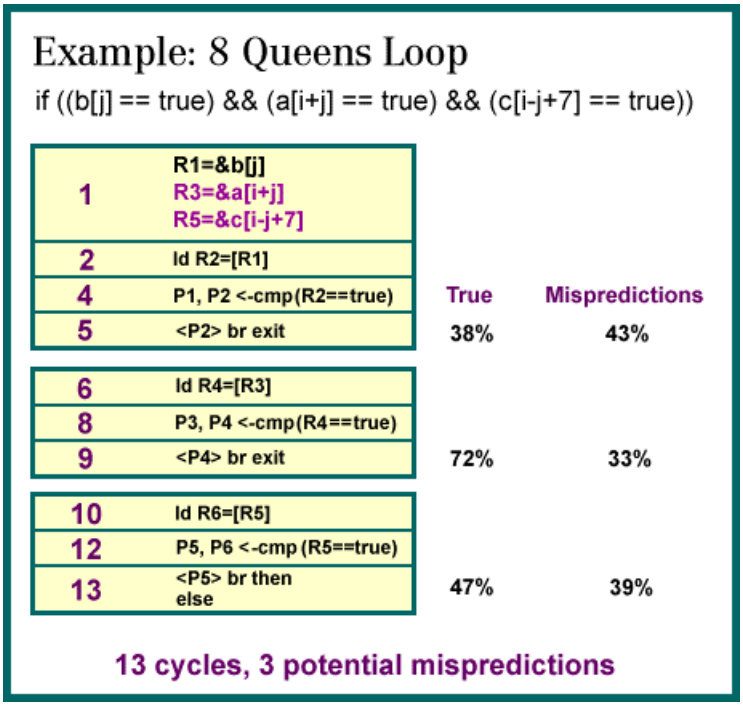
line while they fill out the paperwork at the counter. Similarly, memory latency stalls the processor, leaving it idle until the data arrives from memory. Because memory latency has not kept up with increasing processor speeds, loads (the retrieval of data from memory) need to be initiated earlier to ensure that data arrives when it is needed.

The new 64-bit ISA uses speculation, a method of allowing the compiler to initiate a load earlier, even before it is known to be needed. What if the bank greeter could spot new customers in the bank? If the greeter provides the paperwork for opening an account to all new customers who enter the bank, then customers can finish the paperwork by the time they reach the front of the line. If they don't need the form, they return it to the bank unused. This is comparable to how speculation works. Loads from memory are initiated ahead of time to ensure data is available for use if needed. As a result, the compiler schedules to allow more time for data to arrive without stalling the processor or slowing its performance.

A Coding Example Showing the Concepts Behind the 64-bit ISA

A code-based example can even better illustrate the concepts of explicit parallelism, speculation, and predication used in the new 64-bit ISA. This programming example takes a key statement from the performance benchmark program called "8 Queens Loop." The program is a recursively descending, exhaustive search strategy for finding a way to place eight queens on a chessboard so they cannot attack each other. The program does this by placing a queen on each column of the chessboard such that it cannot be attacked by a queen previously placed either on that row or on the two diagonals. In the following portion of the original code, array "B" checks the row, and arrays "A" and "C" check the two diagonals.

Basically this example uses short-circuit ANDing operators, dividing the problem into three basic blocks and some sequential code. The original code has been broken down into clock cycles of execution, with each clock cycle in a box. Each load has also been charged with two clock cycles of latency. As shown, the original code is definitely sequential -- the instructions are executed in single file, even with a few stops here and there. It is also important to note that, with traditional architectures, this program would take a total of 13 clock cycles to execute and, worse yet, would have the potential for three branch mispredictions.



The second and third instructions in the first block (in green) are the addressing computations for the second and third block. Because these are simple arithmetic calculations, they can be done at any time. They will not cause exceptions, which have a negative impact on the program. The computations have also been moved up and out of the loop -- something that a good compiler could do even better -- so that they have been turned into induction variables, which can be processed simply.

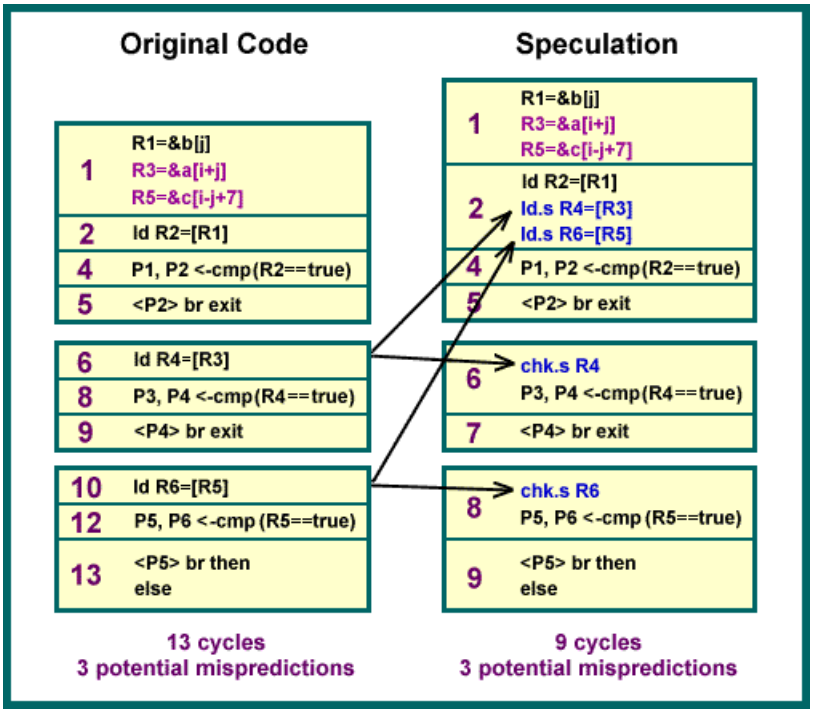
Beyond that, a series of simple operations (load, compare, branch) occurs in sequence. Each instruction is dependent on the previous one, so there is little opportunity to do operations in parallel. Because the loads have been charged with two clock cycle latencies, there is actually a "dead" clock cycle here, since there is nothing else to do between each load and compare.

In the above figure, the branch at the end of the first block falls through to the second block 38% of the time. Similarly, the branch at the end of the second block falls through to the third block 72% of the time, and the branch at the end of the third block falls through 47% of the time.

Applying Speculation to the Example Code

The first transformation applies speculation to remove some of the load latency. Of course, the load in the first block cannot be covered, because the context is insufficient. Potentially, the first block could be moved up higher. However, in the given context, the second load is simply split into a speculative load that is moved above this branch. A check is added to ensure that the load occurred (and without exception), which stays at the home block. This is done for the load in the third block as well. As shown then, the transformation of speculation provides much more parallelism in the code.

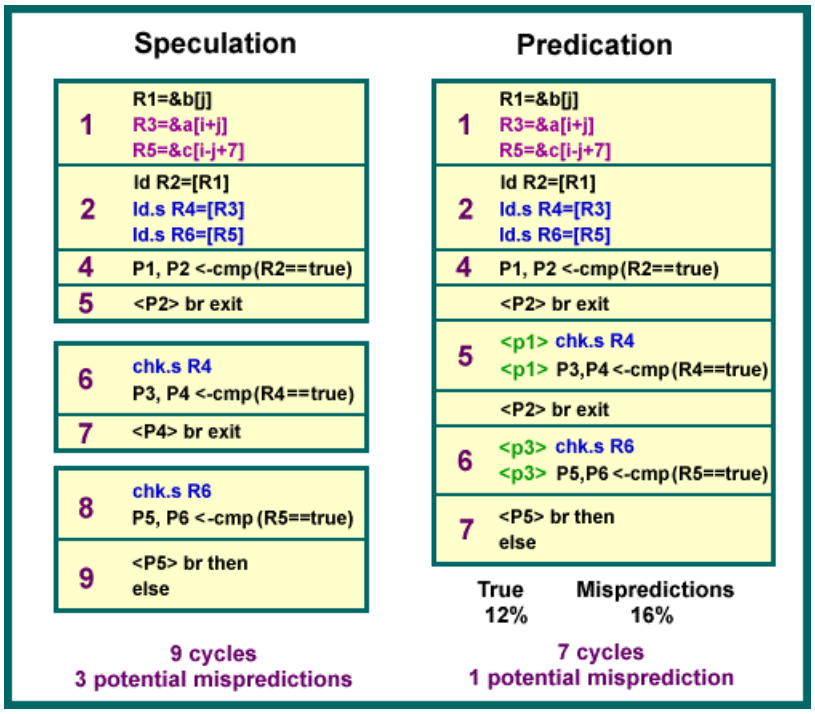
Now if the code is executed on a machine that is capable of executing three memory operations simultaneously, all three loads can be done in parallel. If not, there is still the delay clock cycle, and one of the loads can be inserted, which pairs these instructions. Even in the home block, the compares and the checks can be done in parallel since they are both using the results of the load. Hence, by speculatively executing the loads in this manner, the original 13 total clock cycles are reduced to 9 clock cycles.



Applying Predication to the Example Code

Next, the transformation of predication is applied to the code. In so doing, two of the three branches are eliminated and the code is compacted down into one block, as shown.

A compare was already generating two predicates. So now, instead of branches on the false predicate, execution of the check and the next compare can be qualified on the true predicate. This compare then generates two more results, one of which will qualify the final compare.



The code has therefore been compacted down into one basic block with much more parallelism in it. The total clock cycles required to execute the code have also been reduced by nearly 50%, from 13 to 7. Even more importantly, two of the three mispredictions have been eliminated along with these branches. Originally there was a significant amount of computation, in that the misprediction rates ranged from 33% to 43%, with the potential cost of ten or more clock cycles per misprediction. As the compares and the complex control flow were combined, the critical path was shortened and the number of instructions available for parallel execution increased. In addition, in many cases, particularly with data-dependent branches, an improvement in the branch predictors was also gained.

The Final Results

This simple example showed how applying basic speculation and predication transformations provided in the 64-bit ISA dramatically reduces the critical path in a program and increases the number of instructions that can be executed in parallel. In particular, almost half of the required clock cycles and two-thirds of the potential mispredictions were eliminated from the original "8 Queens Loop" code. An IA-64 based processor would therefore execute this code almost 50% faster than processors using traditional architectures.

Additional Microprocessor Resources

The next generation 64-bit ISA incorporates many innovative features to enable industry leading performance. Because the 64-bit ISA allows the compiler to expose maximum parallelism in the code and explicitly describe it to the hardware, simpler and smaller chip control structures are possible. Space saved on the chip can then be used for additional resources, such as larger caches and many more registers and functional units. These, in turn, supply the processor with a steady stream of instructions and data to make full use of its capabilities, greatly increasing parallel execution and overall performance.

To provide more on-chip resources, Intel's IA-64 based processors capitalize on both the strengths of explicit parallelism and the savings in chip space that the 64-bit ISA provides. IA-64 based processors have massive resources, with 128 general registers and 128 floating-point registers. In contrast, today's RISC based processors typically have only 32 general registers and are therefore forced to use register renaming or some other mechanism to create the resources necessary for parallel execution. In IA-64, the functional units attached to the large register file can also be replicated, making IA-64 inherently scalable over a wide range of implementations. Of course, since replicated functional units increase the machine width, performance can be increased correspondingly. And with the more sizable caches and the many more read and write ports afforded to memory, the speed of IA-64 based processors is no longer limited by the memory latency problems of traditional processors.

Another beneficial effect of IA-64's explicit parallelism is that transistors are used much more efficiently. The 64-bit ISA eliminates the problem of out-of-order dependency logic caused by mispredicted branches.

In addition, the IA-64 instruction format itself enables explicit parallelism, breaking the traditional sequential execution paradigm wherein every instruction is assumed to depend on the previous instruction. The IA-64 instruction format is a 128-bit bundle that contains template bits and three instruction "packs." The template bits define the explicit instruction dependency, flexibly grouping any number of independent instructions (across instruction packs) and specifying the dependency between those groups. That is, the template makes instruction dependency explicit, any number of interdependent instructions can be grouped, and the template directs the dispersal of instructions to functional units. By explicitly scheduling the parallelism in this way, the IA-64 instruction format allows the compiler to expose greater parallelism in the code and express that greater parallelism to the hardware. Similarly, the hardware itself can be simplified, because the dynamic mechanisms required to determine the available parallelism of traditional instruction formats are no longer necessary with the new instruction format of IA-64.

Finally, the hardware is fully interlocked in IA-64 to ensure complete compatibility between machine families, now and into the future.

Summary

Through explicit parallelism, the IA-64 implementation of EPIC technology enables the compiler and hardware to work together efficiently to expose new levels of parallelism and performance. The innovative use of predication and speculation, uniquely combined with explicit parallelism, has allowed EPIC technology to progress well beyond the limitations of traditional architectures, enabling industry-leading performance. These concepts could not be incorporated effectively in existing architectures; they required the creation of a new architecture, IA-64. IA-64 is the first mainstream architecture that was designed from the start to take advantage of parallel execution. IA-64's massive resources, inherent scalability, explicit parallelism, and full compatibility make it the next generation processor architecture for high-performance servers and workstations.